
28 PÁGINAS CON UN DETALLADO TUTORIAL PARA CREAR TU PRIMER JUEGO
250 PESETAS
MICRO
tilemanía
AÑO I
SUPLEMENTO INDEPENDIENTE PARA CURIOSOS QUE
PRIMER VIDEOJUEGO
Número único
tilengine
Diseña un videojuego al estilo “vieja escuela”
ENCONTRARÁS
ANIMACIONES
MECÁNICAS
MOVIMIENTOS
PERSONAJES
EDICIÓN COLECCIONISTA
EDICIÓN COLECCIONISTA
2 DISCOS
6 MEGAS DE INFORMACIÓN
ALTA DENSIDAD
incluye:
GRÁFICOS
CÓDIGO FUENTE
EJEMPLOS
RETRO MANIAC
www.retromaniac.es

MICRO tilemanía

Edita:
RETROMANIAC

Director:
David

Redacción:
Vampirro
Gráficos del juego y correcciones del texto:
Jaime
Director de Arte:
David
Autoedición:
David

Algunas imágenes:
David

Redacción y Publicidad
www.retromaniac.es

Imprime:
Printcolor

Esta publicación se inspira, **a modo de homenaje**, en la primera época de PCMANÍA. ¡Todo un referente!

No solicitidado control O.J.D. (¡pa qué!)

TILEMANÍA no se hace necesariamente solidaria de las opiniones vertidas por sus colaboradores en los artículos firmados. Prohibida la reproducción por cualquier medio o soporte de los contenidos de esta publicación, en todo o en parte, sin permiso del editor.

ISSN: 2171-9969

Este suplemento acompaña el número 11 de RetroManiac para regocijo de todos los amantes de la programación y los videojuegos. ¡Muchas gracias por adquirirlo, de parte de todo el equipo!

Algunas imágenes reproducidas diseñadas por Graphicmama / Freepik, son libres o pertenecen a sus respectivos autores.



suplemento especial | RetroManiac 11

EN ESTE SUPLEMENTO...

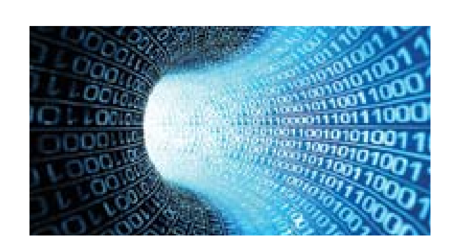
SENTANDO LAS BASES... 4

¿Qué necesito para empezar? Comenzamos nuestro proyecto usando C como lenguaje de programación y Visual Studio Community 2015 como entorno de desarrollo.

Y AHORA EL CÓDIGO ¡ESTO MARCHA!



Ahora que ya tenemos claro cómo colocar los gráficos, es "la hora de la verdad". Toca empezar a introducir la lógica y el control de nuestro personaje.



"PINTANDO" CON TILENGINE 10

Que se vean nuestros gráficos (sprites y fondos) en el programa es responsabilidad directa de Tilengine, descargando al desarrollador de la tediosa tarea de programar la gestión gráfica.



EN PORTADA...



Estableceremos los pasos para crear un juego similar a 'Pang', el exitoso arcade creado por Mitchel Corp, distribuido por Capcom en miles de máquinas alrededor del mundo.

17 CONSEJOS ÚTILES

Tanto si eres nuevo como si tienes tablas en la excitante aventura de crear videojuegos, lee atentamente nuestros consejos tras pasar horas con Tilengine.

MOVIENDO AL PERSONAJE



La "salsa" del juego. Es una de las partes más importantes para que nuestro programa sea atractivo, y es donde descansa gran parte de la lógica de nuestro programa.

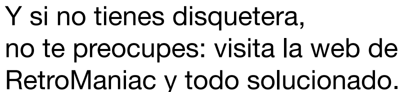




ALTA DENSIDAD



¿Dónde están mis disquetes? Si lo tuyo no es teclear y quieres el resultado rapidito y probar que funciona, no dejes de echar un vistazo a los disquetes que acompañan este suplemento con todo lo que necesitas para crear tu primer videojuego en Tilengine. Y si no tienes disquetera, no te preocupes: visita la web de RetroManiac y todo solucionado.



Este que tienes entre tus manos no es el primer suplemento especial de RetroManiac (ya publicamos otro acerca de la CPCRetroDev 2015), pero si que esperamos que sea el primero de otros suplementos futuros que ayuden a que tú, como lector, te animes a diseñar tu propio videojuego. Hemos elegido un diseño inspirado fuertemente en la legendaria primera etapa de PCmanía, no por capricho, sino como guiño del inigualable trabajo que hicieron en la redacción de esta mítica revista. RetroManiac no se acerca (¡ni lo pretende!) a tan alta cota, aunque apostamos a que te haremos disfrutar con lo que a continuación vas a encontrar: mucho trabajo, esfuerzo, dedicación y respeto por nuestro pasado como usuarios de ordenadores, al servicio en este caso de Tilengine, una serie de librerías algo alejadas de lo que estamos acostumbrados hoy en día en cuanto a la creación de videojuegos, que seguro te va a ofrecer muchos ratos de diversión.


Busca en Facebook "RetroManiac", y hazte amigo.


http://twitter.com/RetromaniacMag





Rememorando los clásicos. Con Tilengine podréis crear juegos de scroll lateral o títulos de carreras a lo Mode 7.

26

www.tilengine.org

SENTANDO LAS BASES

¿Recuerdas los tiempos en los que en las revistas podías encontrar listados de juegos que podías teclear en tu ordenador, para luego jugar? ¿Qué te parecería poder volver a teclear tu propio juego -o copiar y pegar, o lo que quieras- como antaño y, de paso, aprender los rudimentos de la creación de un pequeño videojuego desde cero? Acompáñanos y diviértete aprendiendo a programar tu propio juego retro.

De todas formas, más que simplemente darte un listado y que allí te las apañes tú solo, en RetroManiac hemos pensado que es mejor, si nunca has hecho juegos y te pica la curiosidad, explicarte cómo y por qué se hacen las cosas: cómo pintar sprites en pantalla, cómo animarlos, cómo gestionar las colisiones… Y para la ocasión, hemos escogido hacer un juego que recoja la mecánica básica del 'Pang' de Capcom; es decir, bolas que rebotan en el suelo y que cuando reciben un disparo se dividen en dos más pequeñas, hasta que terminan desapareciendo.

¿Qué necesito para empezar?

En los tiempos de los 8 bits todos los ordenadores incorporaban un intérprete de BASIC de serie -salvo alguna excepción, vale. Pero los principales, todos-, por lo que podríamos decir que era sencillo publicar un listado y que todo el que quisiera pudiera copiarlo. Si tenías un Spectrum, tecleabas el código de Spectrum y listo; no necesitabas nada más.

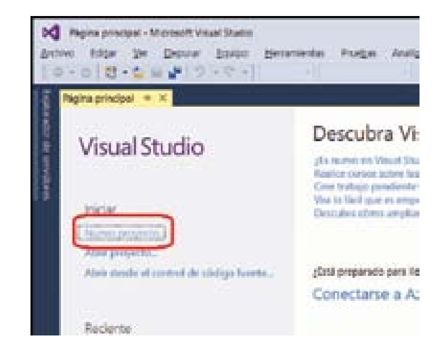
Sin embargo, los ordenadores actuales son diferentes. Tienes multitud de lenguajes, *frameworks*, librerías y miles de utilidades. Nosotros vamos a utilizar Tilengine, la librería gráfica creada por Marc Palacios para la creación de juegos con estética retro, de la que ya hablamos en RetroManiac y de la que, además, hemos publicado un pequeño artículo en nuestro número 11. Aunque puedes utilizar en ella lenguajes como Java o Python nosotros nos decantamos por el más clásico -y eficiente- C/C++. Por lo tanto, lo que necesitarás será como mínimo un editor de textos y un compilador de C/C++. Nosotros hemos utilizado Visual Studio para Windows, el IDE integrado de Microsoft que en su versión Community es gratuito.

También es necesario tener conocimientos de programación. No para teclear el código de nuestro juego, pero sí para poder entenderlo y, lo más importante, poder modificarlo posteriormente a tu gusto. O mejor, hacer uno desde cero. Dado que hemos escogido el lenguaje C/C++, lo más recomendable sería que tuvieras cierta práctica con él para poder centrarte en el juego y no en el lenguaje en sí. En cualquier caso, lo que no es necesario es tener conocimientos en creación de videojuegos.

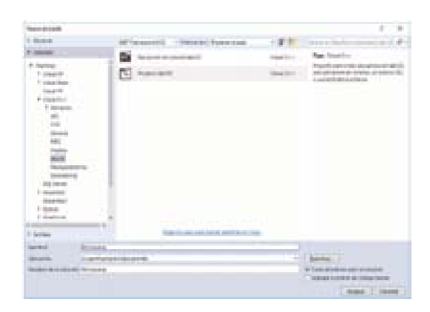
Añadir música y, sobre todo, efectos de sonido, puede enriquecer mucho la experiencia jugable, pero Tilengine es sólo una librería gráfica; añadir sonido implicaría tener que tirar de más librerías y más código, además de tener que generar los efectos de sonido y la música a agregar. En pocas palabras, poner sonido no es imprescindible para el juego y se deja para el lector como "deberes"...

Creación del proyecto

Vamos a usar Visual Studio Community 2015, por lo que todo lo referido al entorno de desarrollo utilizará Visual Studio como punto de referencia. Lo primero que debes hacer es crear un proyecto. Visual Studio funciona por proyectos donde vas metiendo y organizando tus fuentes y ficheros, por lo que lo primero es crear uno. Para ello, debes seleccionar las siguientes opciones (marcadas en rojo):



Ahora escogeréis, entre las plantillas de Visual C++, la de Win32; dentro de la misma, la de Proyecto Win32. A continuación debéis ponerle un nombre e indicar dónde guardar el proyecto:

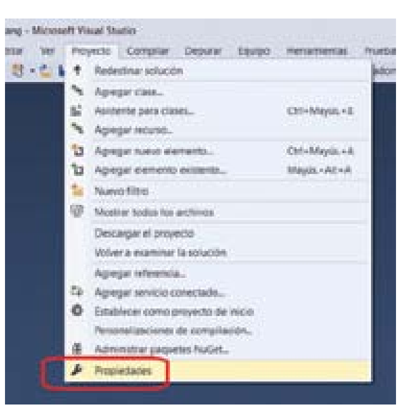


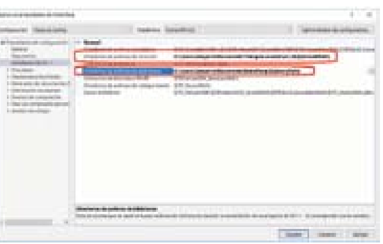
A continuación aparecerá el asistente para este tipo de aplicaciones. Le dais a Siguiente y a continuación seleccionáis Aplicación de consola, quitáis Encabezado precompilado y marcáis Proyecto vacío:



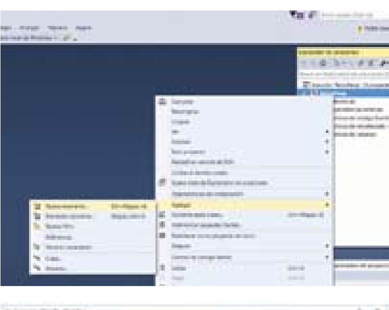
Finalizamos y ahora será necesario configurar el entorno para que funcione con Tilengine. Para ello, hay que copiar las librerías **`Tilengine.lib`**, **`Tilengine.dll`** y **`SDL2.dll`** a la carpeta que nos ha creado el proyecto, en mi caso **`c:\Users\vampirro\documents\RetroPang`**.
Estas librerías estarán en **`<dondeHayasDescomprimidoTilengine>\lib\win32`**. Ahora queda, además de indicarle al proyecto dónde están las librerías que vamos a usar, decirle dónde están los .h que necesitamos para compilar. Volvemos a Visual Studio y entramos en Proyecto>Propiedades:

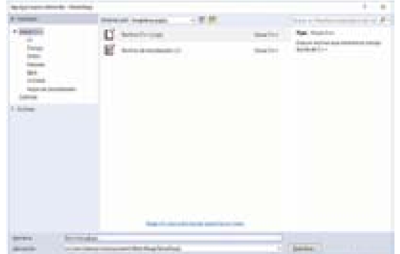
Y ahora debemos añadir de los Directorios de VC++ tanto las librerías como dónde encontrar los .h, tal y como reflejan las siguientes capturas:





Ya tenemos el entorno configurado - ya podemos crear nuestro fichero de código y empezar a teclear. Así que vamos al panel de la derecha (el Explorador de soluciones) y pulsamos sobre “RetroPang” con el botón derecho. Con el menú que nos saldrá pulsamos a continuación en Agregar>Nuevo Elemento y en el cuadro de diálogo que nos aparece seleccionamos dentro de Visual C++ el tipo Archivo C++ (.cpp):
Pues bien, es el momento de empezar



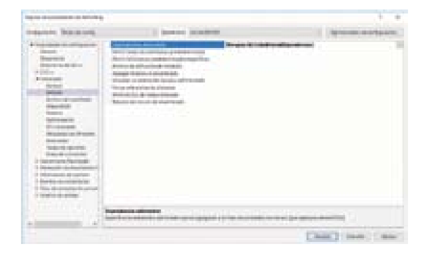


a escribir código. Para asegurarnos que lo hemos hecho bien, podemos escribir el siguiente código:

```
#include <Tilengine.h>

int main(void) {
  TLN_Init(320, 240, 1, 1, 1);
  TLN_CreateWindow(NULL, CWF_
VSYNC);
  while (TLN_ProcessWindow()) {
        // El bucle del juego
  }     // while
  TLN_DeleteWindow();
  TLN_Deinit();
}
```

Al compilar (Control+Shift+B) veremos si ha habido algún error (presumiblemente, ninguno) y al ejecutar (F5) nos saldrá una pantalla en blanco. Por cierto, una cosa más: al ejecutar verás que te sale el error de que no encuentra la librería Tilengine.dll. Para solucionarlo, copia los siguientes ficheros en tu carpeta de Debug, de forma que te quede así:







Diseñando 'RetroPang'

Debido a que 'RetroPang' está fuertemente inspirado en el 'Pang' de Capcom, muchos elementos no necesitarán explicación alguna, pero aun así hay que tener claro qué vamos a hacer. Dada la naturaleza de este juego, 'RetroPang' intentará recoger la jugabilidad básica de 'Pang' pero de una forma limitada para no resultar un proyecto demasiado grande o complejo para el objetivo didáctico que intenta cumplir. Por lo tanto, 'RetroPang' conformará al siguiente diseño:

- El juego contendrá cuatro tipos de bolas, de diferentes tamaños.
- Las bolas se desplazarán de forma horizontal, a la misma velocidad, de un lado a otro de la pantalla, rebotando contra las paredes. Además, irán rebotando en el suelo a diferentes alturas, según su tamaño.
- El jugador estará en el suelo y podrá disparar hasta dos cadenas a la vez, que se moverán de abajo hacia arriba.
- Si una cadena impacta sobre una bola, ésta se dividirá en dos bolas de menor tamaño, que saldrán impulsadas en direcciones opuestas. La excepción a esta regla son las bolas de menor tamaño que, en caso de impactar con una cadena, desaparecerán.
- Si una bola impacta en el jugador, el juego se parará durante un instante y se reseteará para empezar de cero otra vez.
- Desde el inicio irán saliendo bolas del mayor tamaño por la izquierda de la pantalla. Cada pocos segundos saldrá una nueva bola.
- Para evitar que el jugador "se aburra" si limpia de bolas la pantalla, el tiempo de salida entre una bola y la siguiente será cada vez algo menor, hasta alcanzar cierto límite inferior.
- El jugador tendrá tres animaciones: parado, disparando y desplazándose en horizontal.
- Habrá un marcador con los puntos actuales de la partida y otro con la máxima puntuación conseguida hasta el momento - no se almacenará, por lo que cada vez que se cierre el juego el máximo volverá a 0.
- No habrá música ni efectos de sonido.

En Retromaniac ya hemos publicado un artículo explicando el funcionamiento básico de Tilengine, pero volveremos a escribir sobre el tema para volver a recordar los conceptos de la librería.

Tilengine es un motor gráfico diseñado para trabajar "al estilo retro", es decir, no sólo intenta parecer retro sino que está pensado para utilizar técnicas retro como tiles y demás para construir escenarios. No porque se necesite por límite de procesamiento o memoria, sino porque nos gusta y porque nos divierte.

En cualquier caso, es importante saber exactamente cómo funciona. Para empezar, Tilengine te obliga a utilizar paletas limitadas, concretamente con 8 bits de profundidad - es decir, te limita a 256 colores, 255 si utilizas un color como transparente.Lasllamadasbásicaspara tener "algo" corriendo con Tilengine serían:

</>

```
TLN_Init(WIDTH, HEIGHT,
NUM_LAYERS, MAX_SPRITES, NUM_
ANIMATIONS);
TLN_CreateWindow(NULL, CWF_
VSYNC);
// CWF_VSYNC es 60 fps
while (TLN_ProcessWindow()) {
// El bucle del juego
}  // while
TLN_DeleteWindow();
TLN_Deinit();
```



Así de bonito va a quedar el juego una vez terminemos todos nuestros gráficos y echemos a andar el motor del mismo. ¡Tilengine se encargará del resto!

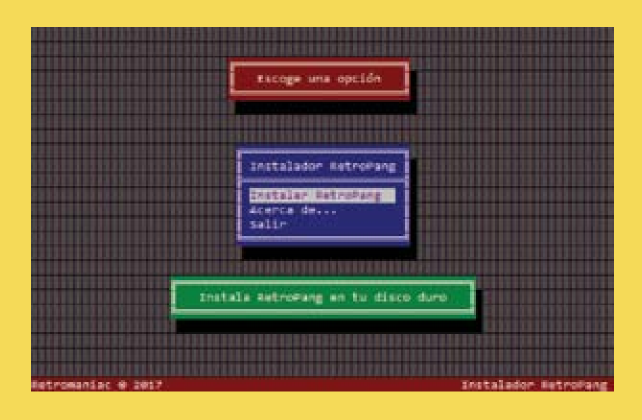
BIENVENIDOS A TILENGINE

Aunque el juego compilado que se incluye en este suplemento de la revista no necesita ser instalado en el disco duro para funcionar correctamente, por temas de velocidad en la carga se recomienda encarecidamente instalarlo. Dado el tamaño del juego, se necesita sólo unos 500 kbytes de espacio en tu disco duro para poder instalar RetroPang.

Los requisitos mínimos para que todo funcione a la perfección son:

- ORDENADOR CON WINDOWS INSTALADO (O MÁQUINA VIRTUAL EQUIVALENTE).
- UNA UNIDAD DE DISCO DE 3.5" DE ALTA DENSIDAD.
- DISCO DURO CON AL MENOS 500 KBYTES LIBRES.

INSTALACIÓN



Lo primero de todo es PROTEGER LOS DOS DISCOS CONTRA ESCRITURA y jamás desprotegerlos. Para ello debes correr la patilla negra hacia abajo de forma que se pueda ver a través de los dos agujeros de cada uno de los disquetes. Pero eso seguro que ya lo sabías, ¿verdad?

El proceso de instalación debe ser de la siguiente forma: INTRODUCIR DISCO 1 EN LA UNIDAD DE 3.5". PASAR A LA UNIDAD DE 3.5" TECLEANDO SU LETRA IDENTIFICATIVA.

Vamos a suponer que tu unidad de 3.5" es la A.

Teclea A: seguido de ENTER, INTRO o RETURN. Luego desde A:> debes escribir: INSTALL
A continuación se abrirá el programa de instalación automatizado, en el que sólo tendrás que seleccionar las opciones que te interesen y esperar a que termine la copia en tu disco duro.



Cuando el sistema termine de copiar los archivos podrás encontrar en tu disco duro una nueva carpeta con sus correspondientes directorios.



No estará de menos que le eches un vistazo a las ventanas de ayuda que encontrarás en el programa de instalación. ¡Si eres persistente quizás encuentres algo interesante!

RETROPANG



Puedes ejecutar directamente el fichero compilado para echar unas primeras partidas a RetroPang y conocer así de primera mano lo que vamos a crear siguiendo las instrucciones del manual.

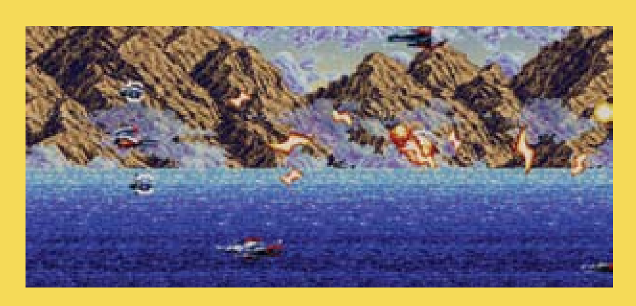
En la carpeta RETROPANG podrás encontrar todo el código fuente tal y como aparece en este suplemento, por si no quieres teclearlo tú mismo (¡con lo divertido que es!). También tienes ya listo el binario para plataformas Windows, que podrías probar inmediatamente para ver el resultado final. Te recomendamos encarecidamente que hagas tus propias modificaciones en el código, mejores el programa, añadas música, etc. ¡Estaremos encantados de que nos mandes el resultado!

TILENGINE

Si no has tenido suficiente con lo que te ofrecemos en el suplemento, también hemos incluido, con permiso de Marc, el código fuente con los ejemplos oficiales de la web de Tilengine en el disco 2, para que aproveches el resto de modos gráficos que ofrece esta potente herramienta, su versatilidad y características. Encontrarás la ayuda y referencias de los comandos en la web oficial: www.tilengine.org

OTRAS PLATAFORMAS

Recientemente Marc ha 'portado' su set de librerías a Raspberry Pi 3 B y dispositivos Android equipados con una CPU ARMv7A, todo un incentivo para esa plétora de desarrolladores independientes que también quieren convertir o crear desde cero sus juegos en estas máquinas.



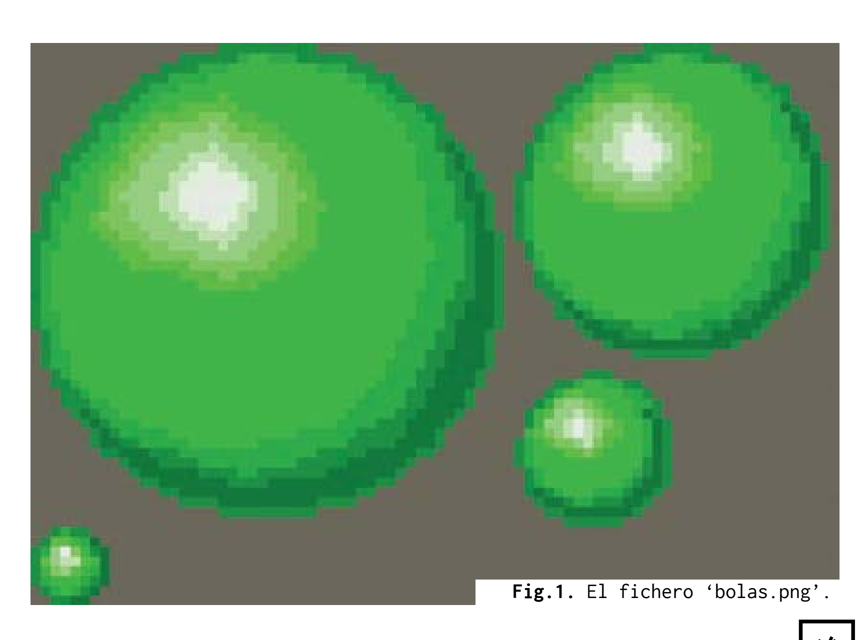
NOTA IMPORTANTE

Este programa de instalación está pensado para ser ejecutado en un sistema operativo tipo Windows. Es probable que si tratas de ejecutarlo mediante máquinas virtuales sin unidades bien mapeadas o empaquetadores-simuladores de un entorno Windows como pueda ser Wine, el sistema te arroje algún tipo de error. Si es así, y no consigues solucionar los problemas con las unidades, careces de una unidad física de discos, etc. no podrás instalar los ficheros tal y como estaba previsto en un principio por nuestro equipo, pero afortunadamente podrás acceder a nuestra web y localizar allí los mismos ficheros que incluimos en los discos, descargarlos y utilizarlos para seguir nuestro tutorial.

Por otro lado, nuestro instalador emula el aspecto que tenían los típicos programas de este tipo durante la primera mitad de la década de los 90. Estudia bien las opciones y no dejes de escudriñar cada rincón de entre sus bits. Es probable que te encuentres con alguna sorpresa agradable mientras te dejas llevar por una agradable sensación de nostalgia.Pero sobre todo lo que queremos es que disfrutes muchísimo, al menos tanto como nosotros lo hemos hecho creando este suplemento, su instalador y el resto de extras.

"PINTANDO" CON TILENGINE

Una vez hemos sentado las bases de nuestro programa, y tenemos claras cuáles son sus mecánicas (interacción, objetivos a cumplir, etc.), puede ser un buen momento para comenzar a escribir las rutinas que nos permitan representar en pantalla todas estas ideas...


Fig.1. El fichero 'bolas.png'.

Pintar sprites en Tilengine

Para dibujar un sprite en pantalla, se necesita por un lado el fichero con formato, por ejemplo, **.png** (¡Ojo! Profundidad de color de 8 bits), y un fichero de texto. La idea del fichero de texto es poder definir distintos fragmentos dibujables dentro del fichero gráfico y darles un nombre. Por ejemplo, para las bolas, se ha utilizado el sprite que podéis ver en esta página con el nombre **'bolas.png' (fig.1).**

Bien, si os fijáis, tenemos cuatro bolas, cada una con un tamaño y una posición concretas dentro del **.png**. Así, el fichero de texto **'bolas.txt'** definirá lo siguiente:

```
bolas48 = 0 0 48 48
bolas32 = 49 0 32 32
bolas16 = 49 33 16 16
bolas8 = 0 49 8 8
```

Es decir, podemos leer que el **sprite bolas48** está en la posición [0, 0] del fichero y ocupa 48 píxeles de ancho y 48 de largo. Y así de forma consecutiva con el resto de sprites.

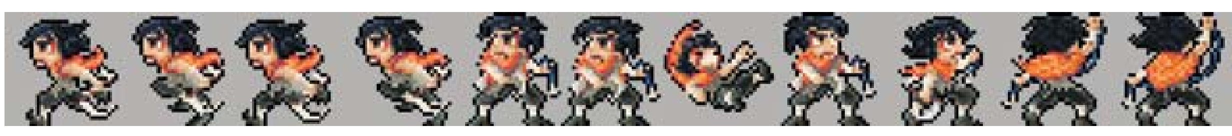
El nombre que le pongamos a cada sprite no es importante de cara a Tilengine, sólo está para hacernos la gestión más simple a nosotros. Lo importante, y la referencia utilizada por Tilengine, es la posición dentro del fichero donde está definido cada sprite. Así, dentro de 'bolas.png' Tilengine comprenderá que hay cuatro sprites, y para utilizarlos nos referiremos a ellos como los sprites del 0 al 3. A este valor lo llamaremos **<numSprite>**.

A nivel de código, se hace de la siguiente manera:



```
TLN_Spriteset sprite;
// La variable donde guardamos los sprites de disco
sprite = TLN_LoadSpriteset(<pathDondeEstéElSprite>);
// Cargamos en la variable el fichero de disco
TLN_ConfigSprite(<spriteID>, sprite, 0);
// Decimos de dónde saldrán los sprites de <spriteId>
TLN_SetSpritePicture(<spriteID>,<numSprite>);
// Le decimos qué sprite concreto es
TLN_SetSpritePosition(<spriteId>, posX, posY);
// Indicamos dónde va en pantalla
```

Siendo **<spriteID>** el identificador por parte de Tilengine del sprite — cuando creamos la ventana le dimos un número de sprites en pantalla; pues uno de esos. Si hemos dicho que va a haber 10 sprites, **<spriteId>** irá de 0 a 9. **<numSprite>** es el identificador de tipo de sprite. Es decir, si



Sprite del protagonista. La animación de los personajes de nuestro juego se representa de esta forma en un archivo en formato PNG. El orden de los gráficos se define posteriormente en un fichero TXT, con sus correspondientes coordenadas.

en el fichero .png y .txt hay 20 sprites diferentes definidos, estos se identifican del 0 al 19; en **<numSprite>** indicaremos cuál de esos 20 es el que vamos a poner en **<spriteId>**.

Pintar animaciones en Tilengine

Para empezar: una animación no es más que una sucesión de sprites en el tiempo. Podemos controlar esto desde el código o podemos utilizar la utilidad de animaciones que nos ofrece Tilengine. En muchos aspectos es exactamente igual que los sprites - es decir, se necesita un fichero gráfico con su correspondiente fichero .txt definiendo dónde está cada sprite y cuánto ocupa. La diferencia esta vez está en un nuevo fichero, '**spriteJugador.sqx**', donde definimos la animación en sí. Usaremos como guía el contenido de **simon.png**, que viene en los ejemplos de Tilengine.

Éste el contenido de '**spriteJugador.txt**':

```
parado = 128 1 24 31
disparo = 161 1 23 31
paso1 = 99 1 27 31
paso2 = 68 1 26 31
paso3 = 35 1 27 31
paso4 = 0 1 32 31
muerto = 188 1 32 32
```

Fijaos que en el documento de texto la posición de parado es la que aparece definida primera, pero en el documento gráfico no está realmente en primera posición. No obstante, desde Tilengine, como es la primera posición definida en el fichero .txt, la dibujaremos accediendo a ella como la posición 0.

Ahora, el contenido de '**spriteJugador.sqx**':

```
<?xml version="1.0"
encoding="UTF-8"?>
<sequences>
  <sequence name="walk" delay="10"
loop="0">
          1,2,3,4,5,6,0
  </sequence>
</sequences>
```

Es decir, es simplemente un formato XML. Aquí lo que hacemos es darle un nombre a la animación -en este caso concreto, "paso"-, indicando que cada fotograma cambiará cada 10 frames. Y la secuencia incluye los sprites definidos en las posiciones 1, 2, 3, 4, 5, 6 y 0 del fichero '**spriteJugador.txt**' (recordemos nuevamente que el primero es el 0); es decir: paso1, paso2, paso3 y paso4... Que a los sprites en el .txt le hayamos puesto este nombre es sólo para aclararnos nosotros, Tilengine ignorará esos literales… aunque sí es importante el nombre de la animación, como veremos ahora. Mediante código, una animación se implementa así:

```
TLN_Spriteset spriteSet;
// Variable para cargar los
sprites en memoria
TLN_SequencePack sp;
// Variable para cargar el fichero de
animación
TLN_Sequence seq;
// Variable para cargar la
animación en sí
spritePersonaje = TLN_
LoadSpriteset(<pathDondeEsteEl.
txt>);
sp = TLN_LoadSequencePack(<ruta del
.sqx>);
```

```
seq = TLN_FindSequence(sp,
<nombreDeLaAnimación>);
TLN_SetSpriteAnimation(<animacion
ID>, <spriteId>, seq, false);
// Aquí hacemos lo que
corresponda, y cuando queramos
quitar la animación:
TLN_DisableAnimation(<animacionID>);
```

En **<nombreDeLaAnimación>** hemos tenido que poner el nombre que tenga en el .sqx - en nuestro caso, "paso"; en **<animacionId>** pondremos un identificador de animación. Cuando creamos la ventana, si os acordáis, teníamos que indicar cuántas animaciones distintas íbamos a usar simultáneamente; si dijimos, por poner un ejemplo, que serían 6, pues tendrá que ser un valor del 0 al 5.

Esto sirve para cargar la animación. Para "pintarla" en pantalla se llama a **TLN_SetSpriteAnimation**, donde asignas a una de las posiciones de sprites la animación creada, porque aunque sea una animación, cada frame de la animación es un sprite, y por lo tanto tiene que ir en una de las posiciones de sprites que hayamos creado.

Una vez asignada la animación a un sprite, ésta se ejecutará, sin necesidad de tocar en ningún sitio más, en la posición que le demos al sprite con **TLN_SetSpritePosition** (ver cómo pintar un sprite para más información) hasta que llamemos a la función **TLN_DisableAnimation**.





Fig. 2. ¡Menudo galimatías! Parece que está todo desordenado, pero en realidad es la forma en la que se construían los gráficos para los escenarios en los videojuegos. En la imagen, los 'tiles' que Sega usó en la primera fase de Sonic para Mega Drive.

Poniendo ladrillos: dibujando con tiles

Realmente, TIlengine, como su propio nombre indica, está pensado para usar tiles. No hay ningún problema en usar grandes escenarios en formato .png e ir moviendo el scroll por ellos, pero Tilengine intenta hacernos programar juegos al viejo estilo, y en el viejo estilo no podías cargar un mapeado de 10 megabytes porque no había máquina que tuviera tanta memoria.

Posiblemente a estas alturas sepas perfectamente lo que es un tile, pero por si acaso… un tile es, por hacer una semejanza, como una pieza de Lego. Las consolas clásicas -y no tan clásicas- tenían esta capacidad: defines pequeñas porciones gráficas y montas escenarios, juntándolas como si de piezas de Lego se tratara. Así se podía conseguir crear vastos mapeados utilizando una cantidad de memoria muy pequeña, pues cada pantalla podía definirse con un puñado de tiles y una referencia a cada uno.

Entre los ejemplos que vienen con Tilengine podemos encontrar el que vamos a utilizar como muestra - seguro que os traerá recuerdos. El fichero se llama **`Sonic_md_fg1.png`** **(fig.2)**.

Con estas piezas se puede construir un muy reconocible escenario de Sonic, pero para ello se necesita, antes de nada, un fichero **`Sonic_md_fg1.tsx`** con el siguiente contenido:

```
<?xml version="1.0"
encoding="UTF-8"?>
<tileset name="sonic_md_bg1"
tilewidth="8" tileheight="8" >
        <image source="Sonic_
md_bg1.png" width="128"
height="104"/>
</tileset>
```

En este fichero indicaremos el tamaño de cada tile (8x8 - lo normal, vamos), dónde está el fichero gráfico con los tiles (**`source="Sonic_md_bg1.png"`**), el tamaño del fichero en píxeles (128x104) y el nombre que le daremos al tileset (**`sonic_mc_bg1`**).

El problema es, ¿cómo se pinta un escenario con esta "paleta" de tiles? Hay diversas herramientas para ello - el autor de Tilengine nos comentó que él había usado Tiled, con el que podemos generar el último fichero que necesita Tilengine para poder trabajar con tiles, **`Sonic_md_bg1.tmx`**. En este caso, el fichero está generado con Tiled, con lo que resulta un "poco" críptico, puesto que el contenido está codificado en base64 y además comprimido, por lo que sin utilizar una herramienta externa como ésta es mejor no molestarse en mirarlo.

Una vez tenemos estos tres ficheros, ¿cómo los utiliza Tilengine? Lo primero es que los tiles, en principio, no están pensados para formar sprites sino fondos y planos de scroll; es una técnica que permite ahorrar mucha memoria al utilizar elementos repetidos de forma secuencial. Por lo tanto, supongamos que queremos hacer un único plano de scroll (¿alguien ha dicho "parallax"? Eso os lo dejo de deberes). El código básico sería tal que así:

```
enum {
LAYER_FONDO,
MAX_LAYER
};
TLN_Init (WIDTH,HEIGHT, MAX_LAYER,
NUM_SPRITES, NUM_ANIMATIONS);
TLN_Tileset tileset = TLN_
LoadTileset ("Sonic_md_bg1.tsx");
TLN_Tilemap tilemap = TLN_
LoadTilemap ("Sonic_md_bg1.tmx",
<nombreDelMap>);
TLN_SetLayer (LAYER_FONDO,
tileset, tilemap);
TLN_SetLayerPosition (LAYER_FONDO,
<posicionX>, <posicionY>);
```

Listo. Hemos cargado en memoria el tileset y el tilemap, y estamos pintando en pantalla la porción visible del mismo según las coordenadas **`[posicionX, posicionY]`**.

INTRODUCIENDO CÓDIGO: ¡ESTO MARCHA!

Quizás la parte que más 'asusta' a aquellos que se enfrentan por primera vez al desarrollo de un videojuego, la programación es ineludible si queremos que nuestro programa sea interesante y proponga al menos un reto interesante al posible jugador.

Bueno, ya hemos hablado suficiente de Tilengine a nivel genérico. ¡Es hora de ponernos manos a la masa! Primero, preparamos el escenario del juego - es decir, la pantalla donde se desarrollará el mismo. Tilengine es muy básico, sólo tiene una pantalla sin scroll. Podríamos pintarla a base de tiles -que, recordemos, es un poco la filosofía de Tilengine y es lo que se solía hacer en su momento, siempre que se podía, para ahorrar memoria- pero nosotros colocaremos el bitmap que aparece en estas páginas **(fig.3)**.

Escenario del juego

Como podéis observar, tenemos una serie de "ladrillos" ornamentales alrededor de una imagen que representa un paisaje de aire japonés, sobre el que se desarrollará la acción. Abajo hay una franja negra donde iremos poniendo tanto la puntuación actual como la máxima.

Bien, con esto son importantes dos cosas. La primera, cómo le decimos a Tilengine que dibuje nuestro bitmap, y la segunda qué espacio de juego nos queda. La pantalla tiene un tamaño de 320x240 píxeles, y los ladrillos tienen 8 píxeles de ancho. Dado que hay un bloque a cada lado, eso hace que el ancho de la pantalla de juego vaya del pixel 8 al 303, y lo mismo a lo alto - ahí irá del píxel 8 al 199. Por lo tanto, además de pintar el fondo declararemoslasconstantesnecesarias para nuestros cálculos futuros de posición:

</>

```
#define TAM_BORDE        8
#define SUELO            199
```

Y ahora, tras crear la ventana, cargamos el bitmap:

</>

```
TLN_Bitmap fondo = TLN_
LoadBitmap("img\\Fondo.png");
TLN_SetBGBitmap(fondo);
```

Bota, bota, la pelota

Es el momento ahora de hablar de una de las cosas más importantes a nivel visual de 'RetroPang': el movimiento y gestión de las bolas que caen. Si el movimiento de éstas es poco natural, o simplemente "aburrido", todo lo demás dejará de ser divertido. Bien, ¿qué necesitamos saber entonces de cada bola?

Aunque ya lo hemos puesto anteriormente, volvemos a mostrar el contenido del fichero descriptor de sprites, **bolas.txt**:

</>

```
bolas48 = 0 0 48 48
bolas32 = 49 0 32 32
bolas16 = 49 33 16 16
bolas8 = 0 49 8 8
```

Vamos ahora a nuestro código. Para poder controlar las pelotas, definimos estos valores:

</>

```
#define HACIA_LA_IZQUIERDA  1
#define HACIA_LA_DERECHA   -1

typedef enum
{
        NINGUNA_BOLA = -1
        , BOLA48
        , BOLA32
        , BOLA16
        , BOLA8
        , MAX_TIPOBOLAS
} tSpriteBola;
```

Con esto le ponemos un nombre a los distintos tipos de bolas, y la estructura que define las diferencias entre bolas es la siguiente:

```
typedef struct {
        double      rebote;
// "Fuerza" del rebote
        int         diametro;
// Diámetro de la bola en píxeles
        int         puntos;
// La cantidad de puntos que da romperla
} t_defBolas;
```

Y ahora que ya hemos definido los valores y estructuras de datos de las bolas, podemos ir rellenando sus valores de configuración. Realmente, para hacerlas cosas bien, estos valores no deberían estar puestos así, "a pelo", en el código, de forma que si quisiéramos cambiar algún valor no tuviéramos que recompilar. Pero ese tipo de cosas os la dejamos a vosotros, así que vamos con nuestro código:

```
t_defBolas defBolas[MAX_TIPOBOLAS];       // Las características de cada tipo de bola
TLN_Spriteset spriteBolas;
  // El spriteSet de las distintas bolas a cargar

void inicializaBolas() {
  spriteBolas = TLN_LoadSpriteset("img\\bolas");

  defBolas[BOLA48].diametro = 48;
  defBolas[BOLA48].rebote   = -11.0;
  defBolas[BOLA48].puntos   = 100;

  defBolas[BOLA32].diametro = 32;
  defBolas[BOLA32].rebote   = -10.0;
  defBolas[BOLA32].puntos   = 150;

  defBolas[BOLA16].diametro = 16;
  defBolas[BOLA16].rebote   = -8.5;
  defBolas[BOLA16].puntos   = 200;

  defBolas[BOLA8].diametro = 8;
  defBolas[BOLA8].rebote   = -7.0;
  defBolas[BOLA8].puntos   = 300;

  for (int i = MAX_SPRITE_BOLAS - 1; i >= 0; i--)
        bolasEnPantalla[i].tipoBola = NINGUNA_BOLA;
} // inicializaBolas
```

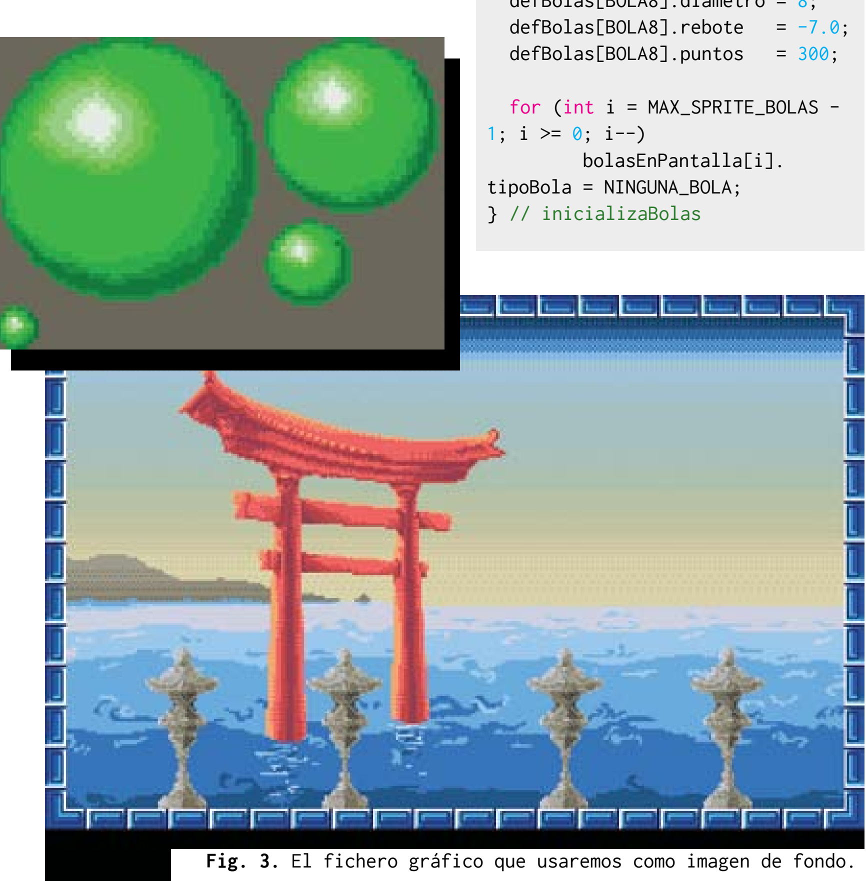


Fig. 3. El fichero gráfico que usaremos como imagen de fondo.

Con esta función le pedimos a Tilengine que cargue los recursos definidos en **'img\bolas.png'** y **'img\bolas.txt'**. Ahí se definen los 4 tipos de bolas, numerados del 0 al 3 - recordemos que, por nuestro tipo enumerado, **BOLA48** vale 0, **BOLA32** vale 1, etc. Como veis, vamos a dar más valor a destruir las bolas pequeñas, pues acabarán siendo más y, al tener un bote menor, serán más difíciles de esquivar. También definimos la altura máxima a la que van a rebotar, aunque eso se verá mejor cuando hablemos de cómo está implementada la gravedad. También aquí se inicializa el array de sprites de bolas a **NINGUNA_BOLA**, que vamos a ver a continuación.

Cada bola será de uno de estos tipos, además de tener el resto de variables necesarias para poder dibujarlas y, sobre todo, saber cómo moverlas:

```
typedef struct {
  int x;
// Posición X del sprite
  int y;
// Posición Y del sprite
  int direccionX;
// HACIA_LA_IZQUIERDA o HACIA_LA_DERECHA
  double velocidad;
// Cuántos píxeles se mueve en cada frame, verticalmente
  tSpriteBola tipoBola;
// Índice de la tabla de bolas, o NINGUNA_BOLA si está vacío
} t_bolas;
```

En estas estructuras todo parece sencillo: una bola será del tipo **t_bolas**, donde guardaremos la posición (x, y) en la que hay que dibujarla, y también el tipo de bola - de 48, 32, 16 u 8 píxeles. También establece la dirección horizontal a la que debe desplazarse (direccionX), y lo bonito de todo está en la velocidad. En cualquier caso, para poder controlar las bolas que hay en pantalla definimos un array así:

```
#define MAX_SPRITE_BOLAS   128
t_bolas bolasEnPantalla[MAX_
SPRITE_BOLAS];
// Las bolas activas en este momento
```

De forma que no podremos tener más de 128 bolas en pantalla - aunque parezca una limitación, creedme, eso son muchas muchas bolas… En la la **(fig.4)** está el código que controla la posición de cada bola.

Esta función recorre el array de bolas buscando aquellos valores que sean dibujables - que no sea **"NINGUNA_BOLA"**. Todas las bolas se mueven siempre a la misma velocidad en horizontal, y lo único que hay que ver es, si tras mover la bola, ha llegado a impactar con un borde. Si es así, se cambia su dirección y ya está.

Como podéis ver, el movimiento horizontal es entero, lo que significa que se moverá 1 píxel por frame, 60 píxeles por segundo. Si quisiéramos hacer el movimiento horizontal de las bolas más lento sería necesario pasar el valor de int a float.

Pero ahora llega la gravedad, que es el código más interesante. En principio es igual de sencillo: la posición de la bola es igual a la que tenía antes, más los píxeles que se indican en **bolasEnPantalla[i]**. velocidad, pero el tema es que la velocidad no es constante. Por lo tanto, la velocidad siempre será:

```
Vi = (Vi-1) + GRAVEDAD
```

```
#define GRAVEDAD                    0.3
#define MOV_HORIZONTAL              1

void mueveBolas() {
  for (int i = MAX_SPRITE_BOLAS - 1; i >= 0; i--) {
        if (bolasEnPantalla[i].tipoBola == NINGUNA_BOLA)
                continue;


        // Movimiento horizontal
        if (bolasEnPantalla[i].direccionX == HACIA_LA_IZQUIERDA) {
                bolasEnPantalla[i].x += MOV_HORIZONTAL;
                if (bolasEnPantalla[i].x >= WIDTH - TAM_BORDE -
defBolas[bolasEnPantalla[i].tipoBola].diametro)
                        bolasEnPantalla[i].direccionX = HACIA_LA_
DERECHA;
        } else {
                bolasEnPantalla[i].x -= MOV_HORIZONTAL;
                if (bolasEnPantalla[i].x <= TAM_BORDE)
                        bolasEnPantalla[i].direccionX = HACIA_LA_
IZQUIERDA;
        }  // else
        // Movimiento vertical
        bolasEnPantalla[i].velocidad += GRAVEDAD;
        bolasEnPantalla[i].y += (int) bolasEnPantalla[i].velocidad;
        if (bolasEnPantalla[i].y >= SUELO - defBolas[bolasEnPantalla[i].
tipoBola].diametro) {
                bolasEnPantalla[i].y -= bolasEnPantalla[i].y - SUELO +
defBolas[bolasEnPantalla[i].tipoBola].diametro;
                bolasEnPantalla[i].velocidad = (double)
defBolas[bolasEnPantalla[i].tipoBola].rebote;
        }  // if


        TLN_SetSpritePosition(i, bolasEnPantalla[i].x,
bolasEnPantalla[i].y);
  }  // for
} // mueveBolas
```

Fig. 4. Código para controlar la posición de las bolas en pantalla.

La esquina superior izquierda de la pantalla es la [0,0], y la inferior izquierda es la [0,240]. Mientras está bajando hacia el suelo, la velocidad debe ser cada vez mayor. Cuando llegue al suelo debe rebotar, es decir, empezar a moverse hacia arriba, por lo que su velocidad se sustituirá por la fuerza de rebote, definida en su tipo de bola, que será un valor negativo. Por lo tanto, en el siguiente frame la bola se moverá -X píxeles: su velocidad vertical más la gravedad. Como la velocidad es negativa, la bola se moverá hacia arriba hasta que, sumándole a cada frame el valor de la gravedad (que la impulsará hacia abajo), llegará a 0 y volverá a ser positiva, por lo que bajará. De esta forma podemos implementar la gravedad sin estar constantemente mirando si la bola va hacia arriba o hacia abajo (**fig.5**).

Otra cosa. En los procesadores modernos, y más tratándose de un juego con un puñado de sprites, el sistema va sobrado de recursos y podemos "desperdiciar" ciclos de reloj sin problemas. Pero en los procesadores de hace más de 20 años, muchos no tenían coprocesadores matemáticos ni tampoco instrucciones para operaciones en coma flotante. Cada ciclo de reloj era precioso - así como cada byte de RAM. Lo que quiero decir es que una suma (o una resta) es menos costosa en tiempo de procesador que una multiplicación, y una multiplicación es menos costosa que una división. Y que comparar si un valor entero

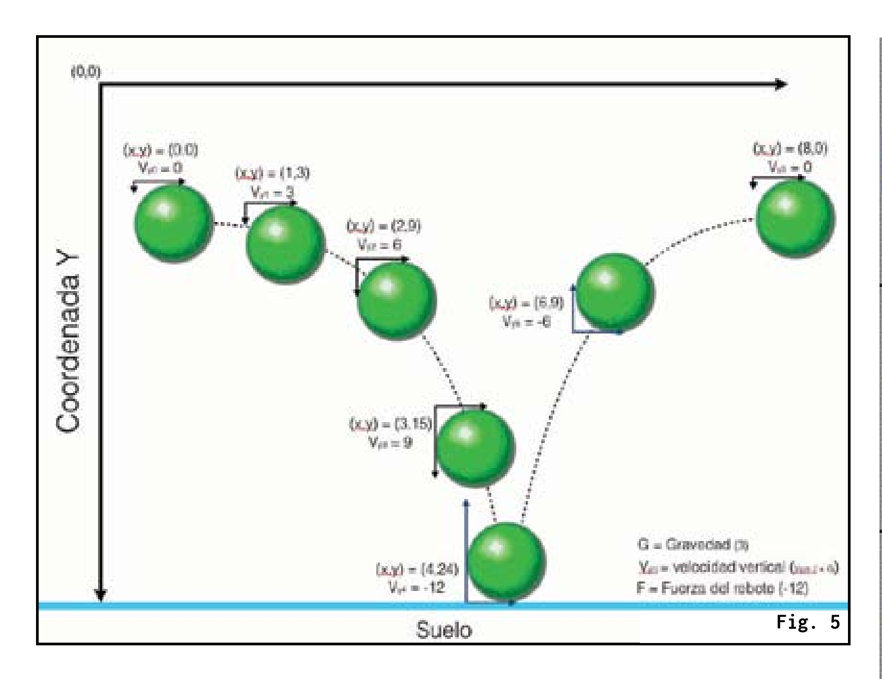


Fig. 5

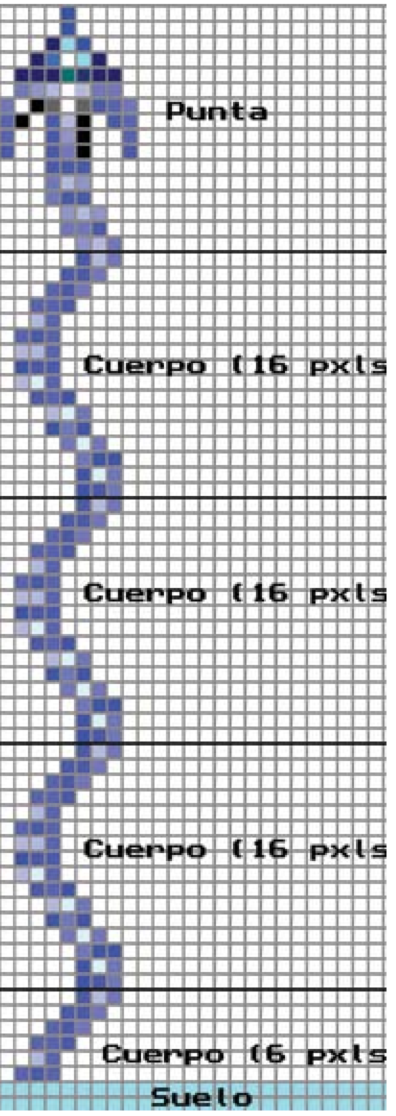


Fig. 6. Construcción de la cadena.

es distinto de cero es más rápido que si es mayor o menor que otro valor. Aunque como digo, si tienes un ordenador de este siglo, irá sobradísimo ejecutando 'RetroPang', pero vamos a intentar no malgastar recursos de procesador aun cuando podamos…

En cualquier caso, como ya hemos calculado las nuevas coordenadas horizontales y verticales de cada bola, llamamos a la función de `Tilengine TLN_SetSpritePosition` para dibujar en pantalla su nueva posición.

Disparando las cadenas

Probablemente la parte más compleja de todo 'RetroPang' sea la animación de las cadenas. A diferencia de otras animaciones basadas en frames como las del personaje jugador, aquí la rotación de sprites se hace mediante código.

Las cadenas se dividen en tres partes; a saber:

1. La punta, o el gancho. Es lo que va lo primero, arriba
2. El cuerpo de la cadena en sí
3. La cola, que es estéticamente igual que el cuerpo pero tiene un tratamiento especial

El problema con la cadena es que no hay un sprite de la cadena como tal sino que, en cierto modo, está tileado. Cuando se dispara, siempre se pinta uno de los frames que pertenecen a la punta —según corresponda—, 0 a n elementos de cuerpo y luego, dependiendo de la posición vertical de la cadena, un sprite de cola o ninguno.

Cada elemento se divide en bloques de 16 píxeles de alto. Cuando se dispara, siempre se dibuja la punta y, según la posición a la que esté en cada momento, se calcula el número de bloques de cuerpo que caben hasta llegar al suelo y se pintan. Pero el problema es el último, ya que hay que calcular cuántos píxeles quedan desde que has pintado el último bloque —ya sea punta o cuerpo— hasta el suelo. Si, por ejemplo, quedan 6 píxeles, no puedes pintar un bloque de 16 del cuerpo sino sólo la porción de 6 píxeles más altos.

Visualmente queda más claro (`fig.6`). Como se puede ver en el gráfico que acompaña al texto, la punta ocupa 16 píxeles de alto y a continuación vienen tres bloques completos de cuerpo de 16 píxeles. Pero del último sólo deben dibujarse 6 píxeles porque, si no, se pintaría por encima del suelo y quedaría un efecto bastante poco estético.

Con las herramientas que proporciona Tilengine, esto se puede hacer de dos maneras. La primera es dibujar un sprite de un píxel de alto, otro de dos, otro de tres… y así hasta uno de 15 (el de 16 sería el normal). Así es como está hecho en el 'Pang' original, y el motivo es porque el hardware sólo maneja tiles de 16x16 píxeles, así que aunque sólo quieras pintar 16x1 píxeles, tienes que hacer un tile de 16x16.

Pero Tilengine no tiene esta limitación; puedes definir tiles y sprites del tamaño que quieras,

por lo que podemos preparar los gráficos de la punta y del cuerpo por separado y luego, en el fichero de configuración que acompaña al .png y define las dimensiones de los sprites, darle el tamaño adecuado. Concretamente, el contenido de **cadena.txt** es:

```
garfio0 = 0 0 9 16
completo1 = 9 0 9 16
fragmento2 = 9 0 9 1
fragmento3 = 9 0 9 2
fragmento4 = 9 0 9 3
fragmento5 = 9 0 9 4
fragmento6 = 9 0 9 5
fragmento7 = 9 0 9 6
fragmento8 = 9 0 9 7
fragmento9 = 9 0 9 8
fragmento10 = 9 0 9 9
fragmento11 = 9 0 9 10
fragmento12 = 9 0 9 11
fragmento13 = 9 0 9 12
fragmento14 = 9 0 9 13
fragmento15 = 9 0 9 14
fragmento16 = 9 0 9 15
```

Realmente esto se repite tres veces para cada frame de animación, pero la idea es la misma. Como puedes ver, cada frame define la punta (el garfio) de 9x16 píxeles; el cuerpo, que sí hay que pintar completo (mismo tamaño, 9x16); y las distintas secciones, según la altura a la que esté la punta y los píxeles que nos queden por rellenar en la última sección antes del suelo.

Una vez explicada conceptualmente la solución al problema de cómo pintar una cadena, vamos a ver el código. Lo primero es preparar algunas medidas para las cadenas:

```
#define MAX_SPRITES_CADENA 12
#define TAMANNO_TILE_CADENA 16
#define Y_INICIAL_CADENA    SUELO -
TAMANNO_TILE_CADENA - 1
#define VELOCIDAD_CADENA 3
#define VELOCIDAD_ANIM_CADENA 5
```

MAX_SPRITES_CADENA se calcula de la siguiente manera: dado que tanto la punta como las porciones de cuerpo tienen **TAMANNO_TILE_CADENA** (16 píxeles), y teniendo en cuenta que desde SUELO (197) hasta arriba (como hay un borde, hasta lo definido en BORDE - es decir hasta la coordenada Y=8) hay 179 píxeles, vamos a pintar 179/16 = 11.18, es decir, hasta 11 porciones enteras de cadena (la punta y 10 más) y una truncada. En total, 12 sprites diferentes, el máximo que puede ocupar una cadena. Y como vamos a permitir **MAX_CADENAS** (dos) a la vez, cuando creamos nuestra ventana lo hacemos con **MAX_SPRITES** como:

```
int MAX_SPRITES = MAX_SPRITE_BOLAS
+ (MAX_SPRITES_CADENA * MAX_
CADENAS);
```

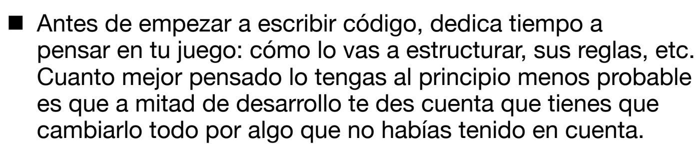
Siendo francos, habría que reservar un sprite también para el jugador, pero bueno, cogeremos uno de las bolas, que 127 aún siguen siendo muchas…

Para manejar las cadenas, prepararemos la siguiente estructura:

```
typedef struct {
  bool activo;
// true = hay que pintarlo; false
= no
  int x;
// Posición x base
  int max_y;
// Hasta qué altura hay que
dibujar la cadena
  int baseFrame;           // El
frame desde el que se empieza a
pintar la cadena
} t_cadena;
```

Por otro lado, hemos dicho que vamos a tener tres frames de animación para la cadena. Así, para inicializarlas, hacemos:

Consejos útiles



Si te ha picado el gusanillo del desarrollo de juegos y quieres crear el tuyo propio, ¡adelante! Aunque desde RetroManiac te damos los siguientes consejos antes de empezar:

- Antes de empezar a escribir código, dedica tiempo a pensar en tu juego: cómo lo vas a estructurar, sus reglas, etc. Cuanto mejor pensado lo tengas al principio menos probable es que a mitad de desarrollo te des cuenta que tienes que cambiarlo todo por algo que no habías tenido en cuenta.
- No desperdicies recursos inútilmente. ¿Alguna vez te has quejado de falta de optimización? Aunque las máquinas actuales vayan muy sobradas moviendo un juego 2D a 320x240 píxeles, acostúmbrate a hacer las cosas bien y no desperdicies recursos porque los hay. Aunque...
- ¡Sin trampas! Cuando se tenía un puñado de kbytes de RAM y menos de 10 MHz para funcionar, había que utilizar toda clase de trucos. Tú no los necesitas. Haz las cosas bien desde el principio, evitando las *ñapas* y te ahorrarás muchos quebraderos de cabeza en el futuro.
- Si vas en serio, usa un control de versiones. Hay muchos: SVN, GIT, el Team Fundation Version Control de Microsoft... Pocas cosas dan más rabia que cargarse algo que funcionaba y no saber dejarlo como estaba.

```
void inicializaCadenas() {
  spriteCadena = TLN_
LoadSpriteset("img\\cadena");

  for (int i = MAX_CADENAS - 1; i
>= 0; i--) {
        cadenas[i].activo = false;
  }  // for

  framesCadena[0] = 0;
  framesCadena[1] = 17;
  framesCadena[2] = 34;
} // inicializaCadenas
```

Es decir, cargamos el gráfico de las cadenas, inicializamos la estructura de control de éstas a "no activo" e indicamos, dentro del fichero cadenas.txt, dónde está la base a pintar de cada frame. La idea es la siguiente:

- Posición definida en **framesCadena** (desde ahora, Base) = punta.
- Base + 1 = cuerpo completo.
- Base + [2..15] = la porción de cuerpo a pintar según los píxeles que falten para llegar al suelo.

Cuando el jugador pulsa el botón de disparo hay que preparar nuestra estructura para la animación de la cadena. Para ello, tenemos la siguiente función:

```
// Inicializa los sprites de animación de una cadena, si los hay
// disponibles (devuelve true). Si no hay disponibles, devuelve false
bool creaDisparo(int x) {
  int i;
  for (i = MAX_CADENAS - 1; i >= 0; i--) {
        if (!cadenas[i].activo) break;
  }      // for
  if (i < 0) return false;
  cadenas[i].activo    = true;
  cadenas[i].x         = x + MITAD_TAM_PER_X;
  cadenas[i].max_y     = Y_INICIAL_CADENA;
  cadenas[i].baseFrame = framesCadena[0];    //Primeraposicióndelosframes
  return true;
} // creaDisparo
```

Por la posición de **Y_INICIAL_CADENA**, siempre se pintarán la punta y al menos un píxel de alto del cuerpo de la cadena. También ponemos siempre el mismo frame de animación de inicio.

Y ahora la parte más complicada: cómo pintar una cadena:
Puesto que puede haber más de una cadena, lo primero es ver cuál nos han pedido pintar, para colocar ahí los sprites que dibujemos. A continuación, desplazamos la cadena hacia arriba el número de píxeles

```
void pintaCadena(int i) {
  int sprite = MAX_SPRITE_BOLAS + (i * MAX_SPRITES_CADENA);
  cadenas[i].max_y -= VELOCIDAD_CADENA;
  int y = cadenas[i].max_y;
  // Comprobamos si hay que cambiar de frames de animación
  if (frame % VELOCIDAD_ANIM_CADENA == 0) {
        cadenas[i].baseFrame++;
        cadenas[i].baseFrame %= FRAMES_CADENA;
  }      // if
  int baseFrame = framesCadena[cadenas[i].baseFrame];
  // En max_y se pinta siempre la cabecera de la cadena, y luego
  // hacia abajo, en bloques de TAMANNO_TILE_CADENA hasta que no caben más
     TLN_ConfigSprite(sprite, spriteCadena, 0);
  TLN_SetSpritePicture(sprite, baseFrame);
  TLN_SetSpritePosition(sprite, cadenas[i].x, y);
  y += TAMANNO_TILE_CADENA;
  baseFrame++;
  for (; y + TAMANNO_TILE_CADENA <= SUELO; y += TAMANNO_TILE_CADENA) {
        TLN_ConfigSprite(++sprite, spriteCadena, 0);
        TLN_SetSpritePicture(sprite, baseFrame);
        TLN_SetSpritePosition(sprite, cadenas[i].x, y);
  }      // for
  // Si los bloques enteros no llegan al suelo, sólo queda un "minibloque"
  int pos = SUELO - y;
  if (pos > 0) {
        baseFrame += pos;
        TLN_ConfigSprite(++sprite, spriteCadena, 0);
        TLN_SetSpritePicture(sprite, baseFrame);
        TLN_SetSpritePosition(sprite, cadenas[i].x, y);
  }      // if
} // pintaCadena
```

definido. También calculamos en qué frame de animación estamos de los tres que tenemos definidos. Como ya hemos dicho, la posición que determina la base de la animación es la punta; luego hay que pintar todos los que se pueda antes de llegar al suelo, y por último hay que calcular cuántos píxeles nos quedan para sumar a la posición del cuerpo completo y ése es el último sprite que hay que pintar - o no, si nos quedan 0 píxeles.

Detección de colisiones

Quizá la parte más peliaguda de un

desarrollo es precisamente ésta, la detección de colisiones. Porque, por muy bien que se muevan las cosas, por muy bien que suene un juego, por muy espectacular que luzca el apartado técnico, si la detección de colisiones falla, todo el juego acaba por venirse abajo.

La forma tradicional de detectar colisiones es mediante *bounding boxes*, o cuadros de detección, pues comprobar si un recuadro está dentro de otro requiere pocos recursos. Dependiendo de cómo sea el sprite dentro del cuadro, se puede hacer tan grande o tan pequeño como se desee. Esto implica que, por un lado, tenemos lo que se ve en pantalla y por otro, el mapeado en memoria de cada elemento, que es lo que realmente hace funcionar el juego. Básicamente, en nuestro caso, dados dos sprites cualesquiera, tendríamos tres casos (**fig.7**):

- Coordenadas cuadrado azul: $[Xa_0,Ya_0]:[Xa_1,Ya_1]$
- Coordenadas cuadrado rojo: $[Xr_0,Yr_0]:[Xr_1,Yr_1]$

Y las condiciones a cumplir serían:

1. Ya_1 siempre debe ser menor o igual que Yr_0.
2. $Xa_0 \leq Xr_0 \leq Xa_1$ (Colisión 1).
3. $Xa_0 \leq Xr_0 \leq Xr_1 \leq Xa_1$ (Colisión 2).
4. $Xa_0 \leq Xr_1 \leq Xa_1$ (Colisión 3).

De todas formas, además de trabajar con Tilengine de esta manera, el motor nos permite una cosa realmente interesante: detección de colisiones al píxel. Para ello, debemos indicar, por cada sprite, si queremos que se active la detección al píxel, de forma que puedes preguntar a Tilengine si un sprite concreto ha colisionado con otro sprite que también tuviera activada la detección de colisiones al píxel.

Por su naturaleza, este recurso nos interesa para saber si una bola cualquiera ha impactado con el sprite del jugador. Si así es —y, en este caso, nos da igual cuál sea; sólo

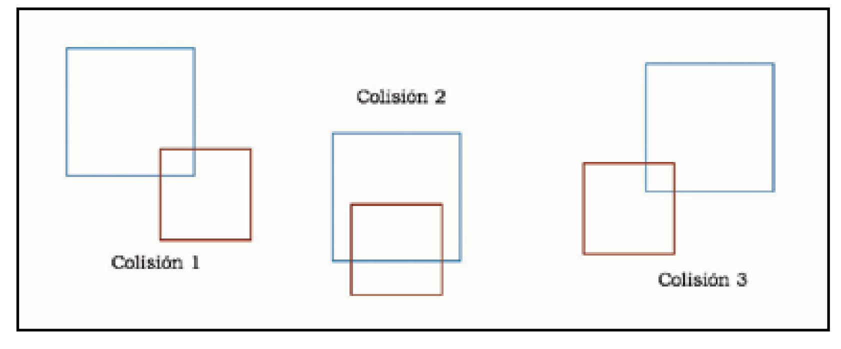


Fig. 7. Los tipos de colisiones.

necesitamos saber que una lo ha hecho—, el jugador ha perdido y se acaba su partida.

En cambio, esta detección no nos sirve realmente para las cadenas, por dos motivos. El primero es que si activamos la detección de colisión en ellas, cuando las disparemos estarían colisionando con nuestro jugador, cosa que no queremos que suceda. El segundo motivo es que si bien podríamos saber que han impactado con una bola, lo que necesitamos es saber con qué bola ha sido, para partirla o destruirla, cosa que el motor de colisiones que incorpora Tilengine no nos permitiría saber.

Por lo tanto, vamos a ver cómo se implementan en 'RetroPang' ambas cosas. Primero, por ser lo más sencillo, la colisión con el jugador. Como hemos dicho, utilizaremos el motor de colisiones de Tilengine, así que lo activaremos tanto para el sprite del jugador como para las bolas, utilizando la función **TLN_EnableSpriteCollision**. Posteriormente, comprobamos si alguna bola ha impactado con el jugador utilizando la función **TLN_GetSpriteCollision(jugador.spriteId)**. Fácil y sencillo, y más que suficiente para saber si algo ha impactado con el jugador.

Veamos ahora el código para los impactos con las bolas. Creamos la siguiente función:

```
int colisionaBolaConSprite(int x,
int tamHorizontal, int maxy) {
  for (int i = MAX_SPRITE_BOLAS -
1; i >= 0; i--) {
        if (bolasEnPantalla[i].
tipoBola == NINGUNA_BOLA)
        continue;
        if (bolasEnPantalla[i].y
+ defBolas[bolasEnPantalla[i].
tipoBola].diametro > maxy &&
                 (
(bolasEnPantalla[i].x > x &&
bolasEnPantalla[i].x < x +
tamHorizontal) ||
    (bolasEnPantalla[i].x +
defBolas[bolasEnPantalla[i].
tipoBola].diametro > x &&
  bolasEnPantalla[i].x +
defBolas[bolasEnPantalla[i].
tipoBola].diametro< x +
tamHorizontal) ||
  (bolasEnPantalla[i].x < x &&
     bolasEnPantalla[i].x +
defBolas[bolasEnPantalla[i].
tipoBola].diametro > x +
tamHorizontal)))
                  return i;
  }     // for
  return NINGUNA_BOLA;
} // colisionaBolaConSprite
```

Siguiendo la nomenclatura que dimos en el gráfico de los cuadrados, indicamos Xr_0, el tamaño horizontal del sprite (Xr_1 = x + **tamHorizontal**), y Yr_0. El algoritmo entonces recorre el array de bolas comprobando si hay alguna cuyas coordenadas estén superpuestas con las del sprite que le han pasado. Si encuentra uno, devolverá su índice, y NINGUNA_BOLA en caso de que nada impacte en el cuadrado del que hemos pasado las coordenadas.

MOVIENDO AL PERSONAJE

Hasta ahora hemos dejado un poco de lado a nuestro protagonista centrándonos en otros elementos del juego, como las bolas o las cadenas, pero tranquilos, que todo llega, y es hora de que nuestro particular personaje cobre vida de una vez por todas, ¡dispuesto a batir récords!

La parte del movimiento del personaje es también compleja, no porque sea difícil en sí, sino porque si bien el personaje sólo se mueve a izquierda y derecha, y dispara, es lo que más cosas distintas permite hacer.

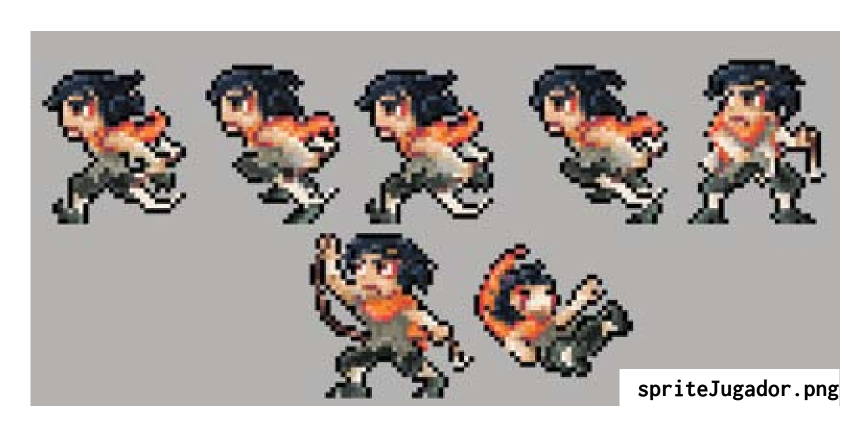
¡Vida, maestro!

También el personaje es lo que más animaciones y trabajo gráfico tiene -en 'RetroPang', se entiende-, ya que no es un sprite estático como puedan ser las bolas, y su estado depende tanto de lo que hace el jugador como de lo de que no hace, o de lo que había hecho antes.

Así, el personaje tiene que:

- Si está disparando, "culminar" su animación sin moverse, salvo que una bola le impacte - fin del juego.
- Si se está moviendo y se sigue pulsando la tecla de desplazamiento, continuar la animación.
- Si se está moviendo y se deja de pulsar la tecla de desplazamiento, pasar al estado "parado".
- Si se pulsa la tecla de disparo, comprobar si el personaje puede disparar. Lo hará si:
    - Ha pasado el suficiente tiempo desde el último disparo.
    - Hay menos de dos cadenas de disparo activas en ese momento.

Por lo tanto, nuestro personaje va a tener cuatro posiciones principales diferentes: "parado", "disparando", "moviéndose" y "muerto". A la hora de moverse, vamos a usar la animación de 4 frames presentada en el apartado sobre pintar animaciones. Si os fijáis, los sprites en movimiento sólo están en una dirección -a la izquierda- mientras que el personaje se puede mover, obviamente, a izquierda y derecha. Tilengine permite girar o reflejar un sprite, de forma que con sólo 4 sprites y una secuencia de animación podemos caminar en las dos direcciones. Volvemos a mostrar el contenido de **spriteJugador.png:**


spriteJugador.png

Bien, dados el fichero .png y el de descripción, crearemos un enumerado que simplifique el código. Según **spriteJugador.txt**, tenemos:

</>

```
enum {
        PARADO
        , DISPARO
        , PASO1
        , PASO2
        , PASO3
        , PASO4
        , MUERTO
};
```

Definiremos también nuestra estructura básica para el control del personaje. Aquí incluiremos el marcador, también, aunque como sólo va a haber un personaje, esto podría dejarse como una variable global. Dado que es el marcador que está haciendo el jugador, aunque esté él sólo, lo asociaremos igualmente:

```
// Estructura para controlar al personaje
typedef struct {
  int spriteId;          // El spriteId del personaje por parte de Tilengine
  int marcador;          // Puntos conseguidos por el jugador
  int x;                 // Posición horizontal del jugador
  int y;                 // Posición vertical del jugador
  int puedeDisparar;     // Frame en el que al personaje se le permite
disparar
} t_jugador;
```

En **spriteId** colocaremos el identificador del sprite del personaje, que será uno de los de nuestro enumerado anterior. Respecto a **puedeDisparar**, lo veremos más adelante. También necesitaremos un par de variables globales: una para saber en todo momento si el jugador se está moviendo o no, y otra para saber si actualmente se halla en posición de disparo. Es decir:

```
bool seEstaMoviendo = false;
// Indica si el personaje está
desplazándose o no
int estaDisparando = 0;
// Indica si el personaje está
disparando actualmente
```

Por otro lado, necesitaremos un par de constantes. La primera, para indicar la velocidad horizontal en píxeles de nuestro personaje, y la segunda para indicar la cadencia de fuego de nuestro personaje. Si pulsas el botón de arriba, teniendo en cuenta que el juego va a 60 fps, el código comprueba cada 1/60 segundos si el botón está pulsado, por lo que por muy rápido que seas, si le das un vez, antes de soltar el botón volverá a comprobar el disparo y aún lo tendrás presionado. Por lo tanto, es necesario que, una vez el juego detecta que has pulsado el botón de disparo, se guarde el frame y te deje disparar hasta que pasen **DELAY_DISPARO** frames. Así pues:

```
#define VELOCIDAD_JUGADOR   3
#define DELAY_DISPARO       15
```

por el de disparo. Y durante dos frames completos -un suspiro: 1/30 segundos- estará “ocupado” con el retroceso del arma y no se podrá mover, por lo que no aceptará ninguna orden de teclado.

Bien, si el personaje no estaba disparando, entonces tendrá que reaccionar a las órdenes de teclado, tanto si las hay como si no. ¿Y qué órdenes admite? Pues disparar



Y ahora, la función (siguiente página) que controla el gráfico del personaje. Por un lado, como lo que tenemos es una animación al caminar, necesitamos saber si lo está haciendo a izquierda o derecha para, si cambiamos la dirección, girar el sentido de la animación.

Lo primero que hace la función es ver si el personaje está disparando, y si lo está, cuánto tiempo lleva. En el momento en que pulsamos la tecla de arriba -la de disparo-, el personaje dispara una cadena, cambiando el sprite que tenga asignado en ese momento



```
// Mueve al personaje en la dirección que le indiquen, o no lo mueve,
// si el jugador no ha pulsado ninguna tecla
void mueveJugador() {
  static bool aIzquierda = true;


  if (estaDisparando) {
        if ((estaDisparando + 2) <= frame)
                TLN_SetSpritePicture(jugador.spriteId, DISPARO);
        else
                estaDisparando = 0;
  } else if (TLN_GetInput(INPUT_UP)) {
        if (jugador.puedeDisparar <= frame && creaDisparo(jugador.x)) {
                jugador.puedeDisparar = frame + DELAY_DISPARO;
                if (seEstaMoviendo) {
                        TLN_DisableAnimation(ANIM_PERSONAJE);
                        seEstaMoviendo = false;
                }       // if
                estaDisparando = frame;
                TLN_SetSpritePicture(jugador.spriteId, DISPARO);
        }       // if
  } else if (TLN_GetInput(INPUT_LEFT)) {
        if (!seEstaMoviendo) {
                TLN_SetSpriteAnimation(ANIM_PERSONAJE, jugador.spriteId,
seqJugador, false);
                seEstaMoviendo = true;
        }       // if
        if (!aIzquierda) {
                aIzquierda = true;
                TLN_SetSpriteFlags(ANIM_PERSONAJE, 0);
        }       // if
        jugador.x -= (VELOCIDAD_JUGADOR * MOV_HORIZONTAL);
        if (jugador.x < TAM_BORDE)
                jugador.x = TAM_BORDE;
  } else if (TLN_GetInput(INPUT_RIGHT)) {
        if (!seEstaMoviendo) {
                TLN_SetSpriteAnimation(ANIM_PERSONAJE, jugador.spriteId,
seqJugador, false);
                seEstaMoviendo = true;
        }       // if
        if (aIzquierda) {
                aIzquierda = false;
                TLN_SetSpriteFlags(ANIM_PERSONAJE, FLAG_FLIPX);
        }       // if
        jugador.x += (VELOCIDAD_JUGADOR * MOV_HORIZONTAL);
        if (jugador.x > WIDTH - TAM_BORDE - TAM_PERSONAJE_X - 6)
                jugador.x = WIDTH - TAM_BORDE - TAM_PERSONAJE_X - 6;
  } else {      // Sprite de personaje parado
        if (seEstaMoviendo) {
                TLN_DisableAnimation(ANIM_PERSONAJE);
                seEstaMoviendo = false;
        }       // if
        TLN_SetSpritePicture(jugador.spriteId, PARADO);
  }     // else
  TLN_SetSpritePosition(jugador.spriteId, jugador.x, jugador.y);
} // mueveJugador
```

Código del movimiento del personaje.

-pulsando arriba-, desplazarse a la izquierda, desplazarse a la derecha y quedarse quieto. En cada posibilidad es importante ver qué estaba haciendo con anterioridad el personaje. Por ejemplo, si el personaje se estaba desplazando hacia la derecha y a continuación se dispara, hay que parar la animación del personaje antes de preparar el sprite y el frame en el que se ha producido el disparo.

Si el personaje se está moviendo, se calcula la nueva posición -siempre sin pasarse de los límites establecidos- y se almacena en la coordenada X del mismo - nuestro mundo es plano. La coordenada Y será siempre la misma, de modo que podríamos haberlo establecido como una constante y no como una variable del personaje. Tras haber calculado qué sprite hay que dibujar y dónde hay que hacerlo, si se está moviendo o no y en qué dirección, si está disparando… se indica a Tilengine qué tiene que dibujar y dónde, mediante **`TLN_SetSpritePosition`**.

¡Muriendo!

El juego se va desarrollando hasta que, en algún momento, una bola impacta con el jugador. Ahí lo que haremos será mantener la imagen congelada durante un segundo, actualizar la puntuación máxima -si procede- y resetear el juego. Para ello, eliminaremos toda la memoria ocupada por los sprites que hubiera, antes de volver a crearlos para la nueva partida. ¿Cosas a destruir?

1. Las bolas. No puede quedar ninguna de la partida anterior.
2. Las cadenas. Si el personaje había disparado antes de morir, hay que eliminar todos los sprites de las cadenas que hubiera en pantalla.

Aparte, el juego pondrá durante un segundo el sprite de muerto, tras el cual volverá a la posición inicial - es decir, personaje centrado en posición de PARADO y con puntuación a 0. Si la puntuación previa era mayor que la puntuación máxima, se sustituye. También se reinicia la dificultad, es decir, la

rapidez con la que van saliendo nuevas bolas.

En código, esta función resolvería el reinicio de la partida cuando impacta una bola con nuestro jugador:

```
#define TIEMPO_REINICIO            1000


///////////////////////////////////////////////////////////////////////
// Deja todo "frito", aunque, eso sí, permitiendo atender los eventos
// de la ventana princial
void implementaRetraso(int tics) {
  // Ahora esperamos el tiempo que indique TIEMPO_REINICIO.
  int tiempoInicial = TLN_GetTicks();
  while (TLN_GetTicks() - tiempoInicial < tics) {
        TLN_ProcessWindow();
  }      // while
} // implementaRetraso


///////////////////////////////////////////////////////////////////////
// Elimina las bolas, pone el marcador a 0 y vuelve todo a empezar
void reiniciaPartida() {
  int i;
  // Lo primero es poner el sprite del jugador a muerto
  if (seEstaMoviendo) {
        TLN_DisableAnimation(ANIM_PERSONAJE);
        seEstaMoviendo = false;
  }      // if
  TLN_SetSpritePicture(jugador.spriteId, MUERTO);
  TLN_DrawFrame(frame);
  frame = 0;
  bolaXframe = DIFICULTAD_INICIAL;
  implementaRetraso(TIEMPO_REINICIO);
  for (i = MAX_SPRITE_BOLAS - 1; i >= 0; i--) {
        if (bolasEnPantalla[i].tipoBola != NINGUNA_BOLA)
                destruyeBola(i);
  }      // for


  if (jugador.marcador > maxPuntuacion)
        pintaMaxPuntuacion(jugador.marcador);
  jugador.x = (int)((WIDTH - TAM_PERSONAJE_X) * 0.5);
  jugador.y = SUELO - TAM_PERSONAJE_Y;
  jugador.puedeDisparar = 0;
  jugador.marcador = 0;
  seEstaMoviendo = false;
  estaDisparando = 0;
  for (i = MAX_CADENAS - 1; i >= 0; i--)
        destruyeCadena(i);
} // reiniciaPartida
```

¡A empezar de nuevo!

Pintando texto

Ahora, una de las cosas más curiosas de una herramienta como Tilengine y que nos recuerda cómo se hacían estas cosas al estilo clásico: ¿cómo pintar texto en pantalla? ¿Qué tipografías tenemos disponibles? Pues… ninguna. La tenemos que crear nosotros. ¿Y cómo se crea una tipografía desde cero? Pues lo primero es definir un fichero donde vamos a poner las letras. Podemos hacerlo de dos formas: poniendo un juego completo de letras -digamos, de 255 caracteres-, o utilizando sólo las letras que vayamos a necesitar. A fin de cuentas, nosotros sólo queremos marcadores, así que con dibujar los dígitos del 0 al 9 sería suficiente, pero vamos a complicarnos un poco más la vida y agregaremos un juego tipográfico completo. Ésta es nuestra tipografía:



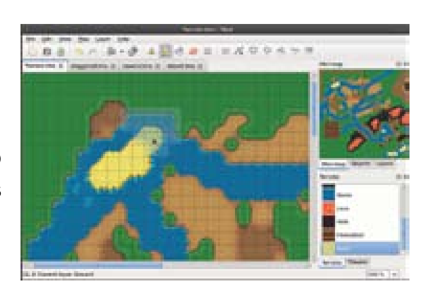
En ella, un carácter ocupa exactamente 8x8 píxeles, por lo que el fichero es de 160x160 píxeles. Para que Tilengine pueda cargar correctamente este documento, necesitamos un descriptor de tiles - es decir, un fichero con extensión .tsx:

```
<?xml version="1.0"
encoding="UTF-8"?>
<tileset name="Font" tilewidth="8"
tileheight="8" spacing="0"
margin="0" tilecount="256">
<image source="font.png"
width="160" height="160"/>
</tileset>
```

En este fichero indicamos que en el código referenciaremos la fuente como Font, y que hay 256 tiles de 8x8 en el fichero font.png, de 160x160 píxeles totales.

Herramientas extra

Dado que Tilengine es una librería gráfica y no un motor como pueda ser GameMaker, Unity 3D o Unreal Engine, es obvio que lo primero que necesitas es un buen editor de texto. Desde el Visual Studio a editores como **UltraEdit**, **SublimeText**, **Notepad++** o similares te pueden servir perfectamente.



De cara a la generación de tiles, una herramienta multiplataforma que te puede venir muy bien es **Tiled**, totalmente compatible con Tilengine. El autor de la librería también recomienda **Spritesheet Packer,** un programa que sirve para 'unir' en un solo gráfico el contenido de varios ficheros independientes. Por nuestra parte, os recomendamos también el uso de algún programa de diseño gráfico como **Grafx2**, **Gimp** o **Photoshop**, o específico para el pixel art, como Aseprite o GraphicsGale.

A nivel de código, para poder añadir texto mediante nuestra tipografía a base de tiles, lo haremos de la siguiente manera: en la primera función, lo que hacemos es inicializar el *layer* de texto, cargando la tipografía en memoria a través de un *tilemap*. Después, dividimos la pantalla en "celdas" de 8x8 píxeles, y por último indicamos que ese layer va a ser "decorado" con los tiles cargados de disco y en la rejilla de 8x8 que hemos creado.

```
// Carga los tiles de la fuente tipográfica
void inicializaTexto() {
  tilesetTexto = TLN_LoadTileset("font.tsx");
  tilemapTexto = TLN_CreateTilemap(HEIGHT / 8, WIDTH / 8, NULL);
  TLN_SetLayer(LAYER_TEXT, tilesetTexto, tilemapTexto);
} // inicializaTexto


// Escribe en pantalla el texto pasado
void pintaTexto (int row, int col, char* texto) {
  Tile tile = { 0, 0 };


  while (*texto) {
        tile.index = *texto + 1;
        TLN_SetTilemapTile(tilemapTexto, row, col, &tile);
        texto++;
        col++;
  }     // while
} // pintaTexto
```

Función para pintar texto.

En la segunda función lo que hacemos es dibujar, a partir de las coordenadas que nos indican, los caracteres indexados por la cadena de texto que nos pasan. Aquí las posiciones no se miden en píxeles sino en las casillas de la rejilla que hemos creado. Así, por ejemplo, para dibujar la puntuación máxima tenemos el siguiente código que utiliza la función **pintaTexto**:

```
// Pinta la puntuación máxima
void pintaMaxPuntuacion(int
puntuacion) {
  char s_marcador[18 + 6 + 1];
  // "Puntos: 000000"
  maxPuntuacion = puntuacion;
  sprintf_s(s_marcador, 18 + 6 +
1, "Mejor: %06d", maxPuntuacion);
  pintaTexto(27, 26, s_marcador);
} // pintaMaxPuntuacion
```

Como vemos, le estamos dando unas coordenadas de 27 y 26, y eso en píxeles es la esquina superior izquierda, mientras que en 'RetroPang' la puntuación máximo aparece abajo a la derecha.



Programa principal

Más o menos ya tenemos definidos o explicados todos los elementos necesarios, así que ahora nos queda definir la lógica principal de nuestro juego. Básicamente, el juego tiene que lanzarnos bolas para que las destruyamos, hasta que nos den. Y cuando esto sucede, reiniciamos el juego. Una vez creadas y cargadas en memoria todas las estructuras del juego, el meollo del mismo consiste en:

</>

```
while (TLN_ProcessWindow()) {
        // Miramos si hay que crear una bola nueva
        if (frame % bolaXframe == 0) {
                creaBola(1, 1, BOLA48, HACIA_LA_IZQUIERDA, 0.0f);

                if (bolaXframe > MAX_DIFICULTAD)
                        bolaXframe -= DIFICULTAD;
        } // if

        // Gestión del movimiento
        mueveJugador();
        mueveBolas();
        mueveDisparos();
        TLN_DrawFrame(frame++);

        colisionaDisparosConBolas();

        sprintf_s(s_marcador, 8 + 6 + 1, "Puntos: %06d", jugador.marcador);
        pintaTexto(27, 1, s_marcador);

        // Gestión de las colisiones
        if (TLN_GetSpriteCollision(jugador.spriteId))
                reiniciaPartida();
  }     // while
```

Programa principal.



Ya tenemos nuestro juego funcionando a pleno rendimiento. Ahora "sólo" caben cambios en los gráficos, en la jugabilidad, añadir objetos para variar la jugabilidad, o hasta un modo para dos jugadores. ¿Te atreves?

Y… ¿ahora qué?

Esperamos que a través de las explicaciones contenidas en este suplemento hayas comprendido cómo utilizar una librería gráfica como Tilengine para poder crear un juego sencillo 2D. Tilengine es realmente potente, pero tampoco es la única librería disponible para hacer juegos 2D con estilo retro. En cualquier caso, te animamos a que experimentes, a que investigues por tu cuenta, ya sea jugueteando con 'RetroPang' o creando tu juego desde cero a partir de las bases que hemos establecido.

Código: Vampirro
Gráficos: Jaime
Tilengine: Marc

EJEMPLOS

BARREL

Scroll lateral con un efecto “tonel” similar al que aparecía en ‘Castlevania IV’ para Super Nintendo.



MODE7

Proyección de un plano 2D en perspectiva para simular un suelo 3D, parecido a ‘Super Mario Kart’.



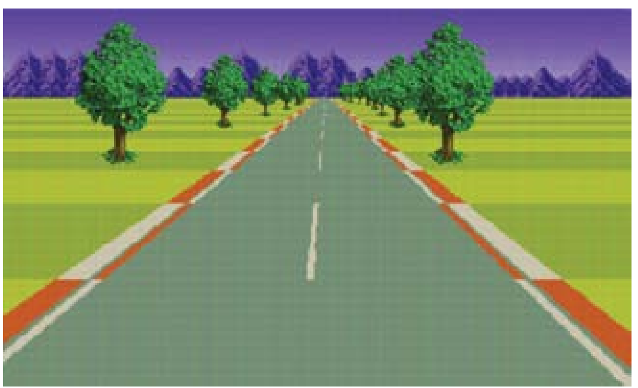
PLATFORMER

Animación de paletas y *linescroll* para simular profundidad con sólo dos capas, como en ‘Sonic’.



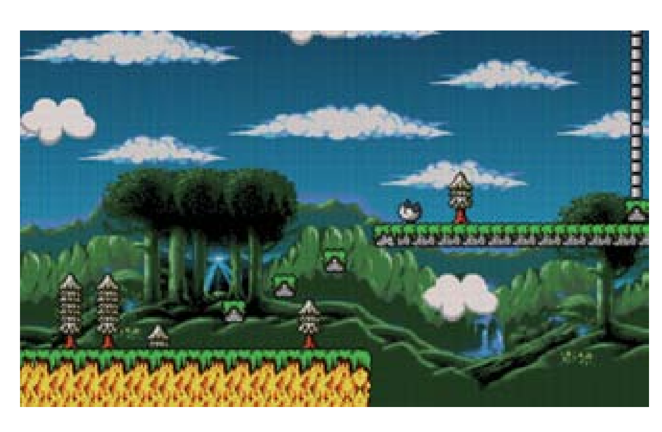
RACER

Escalado de sprites, animación de paletas para el suelo... Seguro que os recuerda a ‘Hang-On’.



SCALING

Combinación de escalado de las capas y composición alfa para las transparencias.



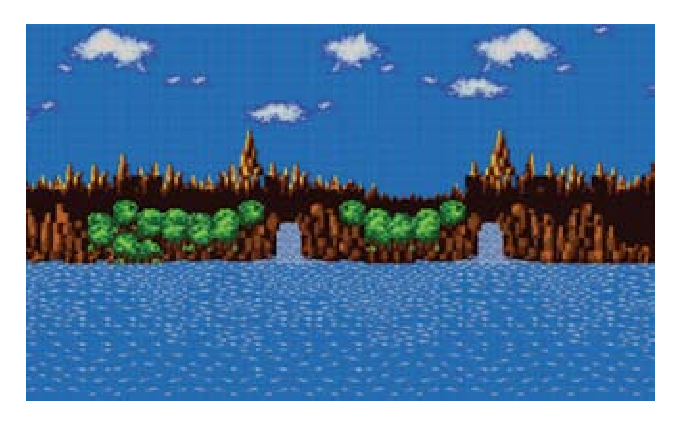
SHOOTER

Animación de paletas y linescroll para simular profundidad utilizando solo dos capas, como en Sonic.



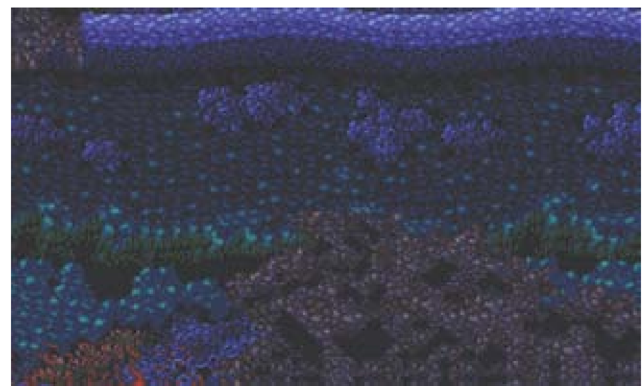
TUTORIAL

Un tutorial muy simple para conocer las bases de Tilengine y cómo empezar a programar con ella.



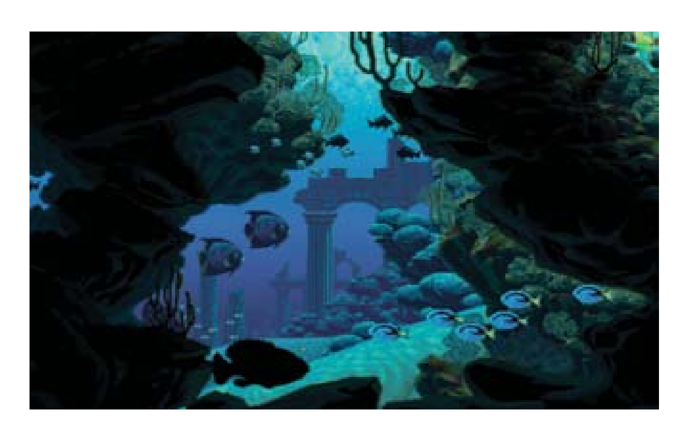
WOBBLE

Un más que interesante efecto para simular la distorsión de la imagen debajo del agua.



COLOR CYCLE

Efectos de "vieja escuela" - animaciones cambiando de posición los colores de la paleta.



Tilengine incluye algunos ejemplos muy clarificadores de lo que puede hacerse con esta excelente librería 2D

SEIZE THE DAY

Impresionante efecto que combina diferentes paletas al tiempo que cambia de posición los colores para simular una transición completa entre la noche y el día. ¡Hay que verlo en movimiento!






RETRO
MANIAC